CS10720 Problems and Solutions

Thomas Jansen

Today: Merge Sort: Algorithm and Analysis
Quick Sort: Partition

March 10th

Important Announcement

Major Oversight in-class test modality
hidden in contents description
'portfolio available during in-class test and exam'

Apologies!

Consequence If you feel disadvantaged
you can re-take a **different** in-class test
(taking into account that you bring material,
consequently no reproduction, more transfer, application)

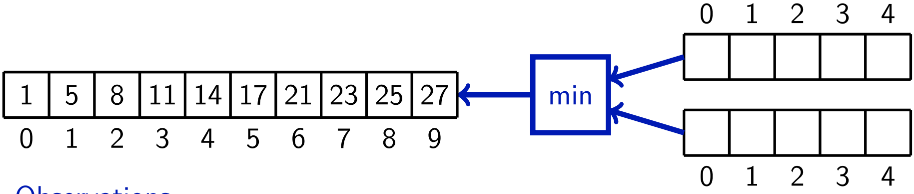
Important If you want to re-take the in-class test
send me email **by Monday (14th March 2016)**

Plans for Today

1. Merge Sort
   - Algorithm
   - Analysis
2. Towards Quick Sort
   - Introduction and Idea
3. Summary
   - Summary & Take Home Message

Remember: Merging Sorted Sequences

An observation leading to an idea for a recursive sorting algorithm
two sorted sequences can be easily merged
into one sorted sequence



Observations

- really simple and fast
- ‘min’ needs to be able to take care of special cases at the end
- extra space required, i. e., not in situ

Idea for a Sorting Algorithm

Remember we can merge two sorted sequences easily and fast
if we allow for extra space
extra array of same size as two input arrays combined

How do we obtain two sorted arrays?

Obviously by means of recursion

Algorithm (Idea) for sorting `int keys[size];`

1. Sort left half of `keys` using the same sorting algorithm.
2. Sort right half of `keys` using the same sorting algorithm.
3. Merge the two sorted halves in a temporary array of equal size.
4. Copy the result back into the input array.

Merge Sort

```
void mergeSort(long *keys, long start, long size) {
   long min1, min2; /* index of first element in each sequence */
   long i; /* position in 'main array' */
   if ( size < 2 )
     return; /* nothing to do for arrays of size < 2 */
   mergeSort(keys, start, size/2); /* sort left half recursively */
   mergeSort(keys, start+size/2, size-(size/2)); /* sort right half recursively */
   /* merge left and right half in tmp; tmp must exist and be large enough!  */
   min1 = start; /* set min1 to smallest element in left half */
   min2 = start+(size/2); /* set min2 to smallest element in right half */
   for ( i=0; i<size; i++ ) { /* in each step copy 1 item to tmp */
     if ( ( min2 >= start+size ) /* right half already empty */
       /* or left half not yet empty and left item minimal */
       || ( ( min1 < start+(size/2) ) && ( keys[min1] <= keys[min2] ) ) )
       tmp[i] = keys[min1++]; /* copy from left half and adjust its index */
     else
       tmp[i] = keys[min2++]; /* copy from right half and adjust its index */
   }
   /* copy resulting sorted sequence over the original input */
   for ( i=0; i<size; i++ )
     keys[start+i] = tmp[i];
}
```

Analysis of Merge Sort

Observation merge sort is easy to implement and elegant
but merge sort is not in situ
since it needs extra space n for input of size n
much more than $O(\log n)$ allowed for in situ sorting algo.

Is it efficient? Rememeber $\Theta(n \log n)$ is what we want

```
void mergeSort(long *keys, long start, long size) {
 long min1, min2; /* index of first element in each sequence */
 long i; /* position in 'main array' */
 if ( size < 2 )
   return; /* nothing to do for arrays of size < 2 */
 mergeSort(keys, start, size/2); /* sort left half recursively */
 mergeSort(keys, start+size/2, size-(size/2)); /* sort right half recursively */
 /* merge left and right half in tmp; tmp must exist and be large enough!  */
 min1 = start; /* set min1 to smallest element in left half */
 min2 = start+(size/2); /* set min2 to smallest element in right half */
 for ( i=0; i<size; i++ ) { /* in each step copy 1 item to tmp */
   if ( ( min2 >= start+size ) /* right half already empty */
     /* or left half not yet empty and left item minimal */
     || ( ( min1 < start+(size/2) ) && ( keys[min1] <= keys[min2] ) ) )
     tmp[i] = keys[min1++]; /* copy from left half and adjust its index */
   else
     tmp[i] = keys[min2++]; /* copy from right half and adjust its index */
 }
 /* copy resulting sorted sequence over the original input */
 for ( i=0; i<size; i++ )
   keys[start+i] = tmp[i];
}
```

recursion

merging

copying $\Theta(n)$ ✓

Analysis of Merging

```
/* merge left and right half in tmp; tmp must exist and be large enough!  */
min1 = start; /* set min1 to smallest element in left half */
min2 = start+(size/2); /* set min2 to smallest element in right half */
for ( i=0; i<size; i++ ) { /* in each step copy 1 item to tmp */
  if ( ( min2 >= start+size ) /* right half already empty */
    /* or left half not yet empty and left item minimal */
    || ( ( min1 < start+(size/2) ) && ( keys[min1] <= keys[min2] ) ) )
    tmp[i] = keys[min1++]; /* copy from left half and adjust its index */
  else
    tmp[i] = keys[min2++]; /* copy from right half and adjust its index */
}
```

Observations

- run time is dominated by loop over n items
- everything inside the loop has run time $\Theta(1)$
- total run time is $\Theta(n)$ ✓

Analysis of Merge Sort

```
void mergeSort(long *keys, long start, long size) {
  long min1, min2; /* index of first element in each sequence */
  long i; /* position in 'main array' */
  if ( size < 2 )
    return; /* nothing to do for arrays of size < 2 */
  mergeSort(keys, start, size/2); /* sort left half recursively */
  mergeSort(keys, start+size/2, size-(size/2)); /* sort right half recursively */
  /* merge left and right half in tmp; tmp must exist and be large enough!  */
  min1 = start; /* set min1 to smallest element in left half */
  min2 = start+(size/2); /* set min2 to smallest element in right half */
  for ( i=0; i<size; i++ ) { /* in each step copy 1 item to tmp */
    if ( ( min2 >= start+size ) /* right half already empty */
      /* or left half not yet empty and left item minimal */
      || ( ( min1 < start+(size/2) ) && ( keys[min1] <= keys[min2] ) ) )
      tmp[i] = keys[min1++]; /* copy from left half and adjust its index */
    else
      tmp[i] = keys[min2++]; /* copy from right half and adjust its index */
  }
  /* copy resulting sorted sequence over the original input */
  for ( i=0; i<size; i++ )
    keys[start+i] = tmp[i];
}
```

recursion

merging $\Theta(n)$ ✔

copying $\Theta(n)$ ✔

See run time of `mergeSort(long *keys, long start, long size)` is time needed for recursion plus $\Theta(\text{size})$ for merging and copying

A bit more formal Let $T(n)$ be run time of merge sort for input size n

$$\begin{aligned} T(n) &= \text{'time for recursion'} + \Theta(n) \\ &= \text{'time for mergeSort(left half)'} \\ &\quad + \text{'time for mergeSort(right half)'} + \Theta(n) \\ &= T(n/2) + T(n/2) + \Theta(n) = 2T(n/2) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

You either believe this or look into maths.

Extra Information: Run Time Analysis

We know $T(n) = 2T(n/2) + \Theta(n)$

Observe
$$
\begin{aligned}
T(n) &= 2T(n/2) + \Theta(n) \\
&= 2 \cdot (2T(n/4) + \Theta(n/2)) + \Theta(n) \\
&= 4T(n/4) + 2\Theta(n/2) + \Theta(n) \\
&= 4 \cdot (2T(n/8) + \Theta(n/4)) + 2\Theta(n/2) + \Theta(n) \\
&= 8T(n/8) + 4\Theta(n/4) + 2\Theta(n/2) + \Theta(n) \\
&= 16T(n/16) + 8\Theta(n/8) + 4\Theta(n/4) + 2\Theta(n/2) + \Theta(n) \\
&\dots \\
&= 2^k T\left(n/2^k\right) + \sum_{i=0}^{k-1} 2^i \cdot \Theta(n/2^i) \\
&= nT(1) + \sum_{i=0}^{(\log_2 n)-1} 2^i \cdot \Theta(n/2^i) \\
&= \Theta(n) + \sum_{i=0}^{(\log_2 n)-1} 2^i \cdot \Theta(n/2^i) \\
&= \Theta(n) + \sum_{i=0}^{(\log_2 n)-1} \Theta(n) = \Theta(n) + \Theta(n \log n) \\
&= \Theta(n \log n)
\end{aligned}
$$

Recursion Depth of Recursive Merge Sort

We have $$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &\cdots \text{ (a bit of maths or `magic'; see previous slide)} \\ &= \Theta(n \log n) \end{aligned}$$

If you were following the proof. . .

Observation recursion depth is $k = \log_2 n$

If you preferred to just believe. . .

Fact recursion depth is $k = \log_2 n$

Consequence additional space requirement is $n + \log_2 n = \Theta(n)$

Summary Merge Sort

Theorem

Merge Sort sorts n items in an array in time $\Theta(n \log n)$ using extra space $\Theta(n)$.

Example

```
mergeSort([3, 4, 7, 8, 9, 10, 12, 14], 0, 8)
```

Remember: Recursion

- recursion allows for simple, natural algorithms using pattern
  1. Check for trivial solution and stop if possible.
  2. Do something to make the problem smaller.
  3. Use the function itself to solve the smaller problem.
- recursion uses more spaces than is obvious
  due to space on call stack
- recursion requires care that call depth does not get too large
- when (obvious) iterative solution is available
  it is almost surely better
- Merge Sort can sort n items in time $\Theta(n \log n)$ with additional space n and call depth $O(\log n)$

Recursive Sorting

Remember Merge Sort for `int keys[size];`

1. If size < 2 do nothing and stop.
2. Sort left half of `keys` using Merge Sort.
3. Sort right half of `keys` using Merge Sort.
4. Merge the two sorted halves in a temporary array of equal size.
5. Copy the result back into the input array.

Now another recursive sorting algorithm
based on a completely different idea

Merge Sort	Quick Sort
1 Sort parts recursively.	1 Partition into two parts.
2 Merge parts.	2 Sort parts recursively.

Partitioning

What do we mean by 'partitioning the input into two parts'?

Definition Pick one input element (called pivot element)
re-arrange input such that
every element $<$ pivot is left of pivot and
every element $>$ pivot is right of pivot
(elements $=$ pivot can end up in either part but only in one)

Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
32	18	3	9	47	11	1	8	5	6	17	2	31	38	52	12	13	4

↓

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
4	13	3	9	12	11	1	8	5	6	2	17	31	38	52	47	18	32

Towards Partitioning

Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
32	18	3	9	47	11	1	8	5	6	17	2	31	38	52	12	13	4

↓

4	13	3	9	12	11	1	8	5	6	2	17	31	38	52	47	18	32
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Observations

- partitioning does not mean sorting
  (means it could be much easier to do)
- pivot element might have to change its place
- after partitioning pivot element has its correct place in sorted sequence

Partitioning: The Simple Case

Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
19	12	3	2	18	21	15	17	27	45	50	41	31	38	52	32	43

Idea start looking for incorrectly placed items
left and right of the pivot element and swap if found

Observation simple and efficient

But what if we find an incorrectly placed item
on only one side of the pivot element?

Partitioning: The One-Sided Case

Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	5	1	7	19	42	27	58	30	8	33	10	16	24	13	11	83

Idea start looking for incorrectly placed items
left and right of the pivot element and swap if found

Observation need to consider pivot element itself ‘incorrectly placed’
otherwise it may not reach its correction position

Observation need to keep track of position of pivot element
which can change if it gets swapped

Will this always work?
What if there are multiple items with equal keys?

Partitioning: The Case With Equal Keys

Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
7	9	18	24	6	24	31	52	38	81	63	72	99	24	36	54	25

Idea start looking for incorrectly placed items
left and right of the pivot element and swap if found

Observation need to consider pivot element itself 'incorrectly placed'
otherwise it may not reach its correction position

Observation need to keep track of position of pivot element
which can change if it gets swapped

Observation need to force to move on
to avoid being caught in endless loop

Partitioning

Remember

- start looking for incorrectly placed items left and right of the pivot element and swap if found
- need to consider pivot element itself 'incorrectly placed' otherwise it may not reach its correction position
- need to keep track of position of pivot element which can change if it gets swapped
- need to force to move on to avoid being caught in endless loop

Setting work in array `int keys[size]`
have initially `left` denote the leftmost index
have initially `right` denote the rightmost index
have pivot give index of pivot element

Summary & Take Home Message

Things to remember

- analysis merge sort
- efficient partitioning

Take Home Message

- Recursive algorithms can be natural and simple.

Lecture feedback `http://onlineted.com`